NASA-CR-200843

FINAL
7N-62-CR
OCIT
43897

The Rationale of Distributed Rendering Using OpenGL[1] in a Parallel CFD Environment

Dr. Michael L. Stokes[2],
Dr. Ming-Hing Shih[3]

ABSTRACT

The advent of parallel and distributed processing has lead researchers to seek new and innovative approaches to visualization of large memory, time dependent problems such as those encountered in Computational Fluid Dynamic(CFD) applications. The current approach uses a concept whereby graphic calls can be issued at remote sites in a distributed parallel environment and appear within a single window on a workstation. This approach eliminates the need for explicit data migration to a centralized site for purposes of visualization. This paper describes a prototype implementation of a distributed CFD simulation system using the concept of distributed rendering. The rationale of this design is compared to an approach in use at NASA Ames which encapsulates the graphics entities into extracts[1.] which are then transferred to a remote client process and rendered locally.

Introduction

Recent technical advances in and availability of parallel computer architecture has spurned much development with the CFD community. As a result of more available memory, simulation sizes can routinely accommodate four million points or more. Since most currently available visualization software packages such as FAST[2.], and PLOT3D[3.] rely only on local memory, the entire domain of interest must fit within one computer memory image, a feat difficult for even the most advanced large memory workstation. This obviously restricts the researcher to observe only a subset of the domain at any one time.

Since the availability of clustered workstations is often a scheduled resource, the researcher must make optimal use of the time window allotted. The researcher no longer has the option of submitting a long running job requiring the use of many processors, just to find out that there was a small error in boundary or run-time parameters that invalidated the results of the job. To minimize waste in computational resource, it was necessary to monitor the early development of the fluid field to validate boundary conditions and run-time parameters, correct inputs, then quickly get the job back into queue. To facilitate the management and transport of data that would have to be managed, a vastly different model than the current monolithic model would be required.

This work is also of interest to provide visualization in a object based simulation environment which is currently under investigation at the Engineering Research Center(ERC) at Mississippi State University. In this environment, objects such as computational grids are distributed on remote processors using CORBA(Common Object Request Broker Architecture[4.]). In an object oriented environment, the appearance of an object on the display is defined within the definition of the object

1. This work was supported by the NASA grant NAG 2-824, NASA Ames Research Center, Kristina D. Mecili contract monitor.
2. Research Faculty member Mississippi State University/NSF Engineering Research Center
3. Post Doctoral position Mississippi State University/NSF Engineering Research Center

FINAL

itself, typically a drawing routine or *method* defined by the object. However, if this routine is executed on a remote processor, the mechanism it uses must be able to display the results on a remote display. The current method would provide such a capability and greatly simplify the problem of coupling distributed processing and distributed visualization.

For a CFD simulation system, a critical design goal was that the visualization component must be as non–obtrusive as possible to the performance and operation of the parallel system. This translates into the requirement that the performance of the system not be seriously degraded as a result of the visualization, and that the visualization system be "detachable" from the system when the visualization was not needed. Similarly, it would be an advantage if the visualization could be "attached" to a running process to facilitate diagnostics if the temporal residual of the field convergence was large or misunderstood.

The current model was derived on the basis that processing should occur on the processor that contains the data. This is a simple extension of the object–oriented concept of domain decomposition in which each processor is mapped to a computational sub–domain and is responsible for all processing on that sub–domain, including visualization. X Windows, for example, provides some insight as to how this could work by its ability to have a local process assign it's output to a remote display. In the current model, however, we require that multiple remote processes map their output to a single window. While not currently exploited in many applications, X Windows does provide this capability in that the X Server considers the *screen* and *pixmap* as sharable resources and thus allows processes on remote computers to use these as *drawables*. On the UNIX operating system, OpenGL is supported through extensions to the X Window protocol using similar concepts that will discussed in a following section.

Architecture of the Field Solver

The field solver chosen for this work was the 3D structured NPARC[5.] Navier–Stokes solver developed originally at AEDC, and now maintained by AEDC and NASA Lewis. This code was chosen because it was publicly available(distribution C) and because it supported multi–block operation. Though this version did not support parallelization, it was reasoned that the conversion would be straight–forward due to its support for out of memory multi–block operation, as was indeed the case. To make the parallelization as general as possible and thus keep portability high, PVM[6.] was chosen as the mechanism by which information was exchanged between the various blocks. This chose allows everything from personal computers to supercomputers be used as remote processors in the parallel system.

The main program of the NPARC code was slightly modified to facilitate use of shared memory for use by the visualization process. This was simplified in that the NPARC code allocates a large work array (in **FORTRAN**) from which addresses are assigned to sub–arrays through **SUBROUTINE** argument calls. The array declaration was replaced by a call to a small C routine that created a shared memory whose address was then stored in a **POINTER**. This memory is then accessible to both the CFD solver and the visualization process. However, both processes must exercise caution that one process does not attempt to read an address that is currently being written by the other process(which usually causes both tasks to abort). To prevent this, IPC semaphores were created along with a small set of C routines callable from **FORTRAN** which protect critical regions of the code from ill wanted access from multiple processes running on the same processor.

Architecture of Visualization

To support visualization, the solver process on each remote processor creates a child process that is responsible for the visualization on the sub–domain. To facilitate the sharing of information between the solver(parent) process and the visualization(child) process, the IPC (Inter–Process Communication) library was used to provide shared memory as well as semaphores to control read and write access to the shared memory. This design has several salient features: Since memory is shared, the overhead of moving information between the solver and visualization processes is minimized and requires only one data image exist on that processor. Secondly, since the visualization is contained in a separate process, the memory taken up by that process disappears as soon as the process is no longer needed. Similarly, this architecture allows the visualization process to be attached to a running simulation to monitor the on–going process. This process is illustrated in the following figure.

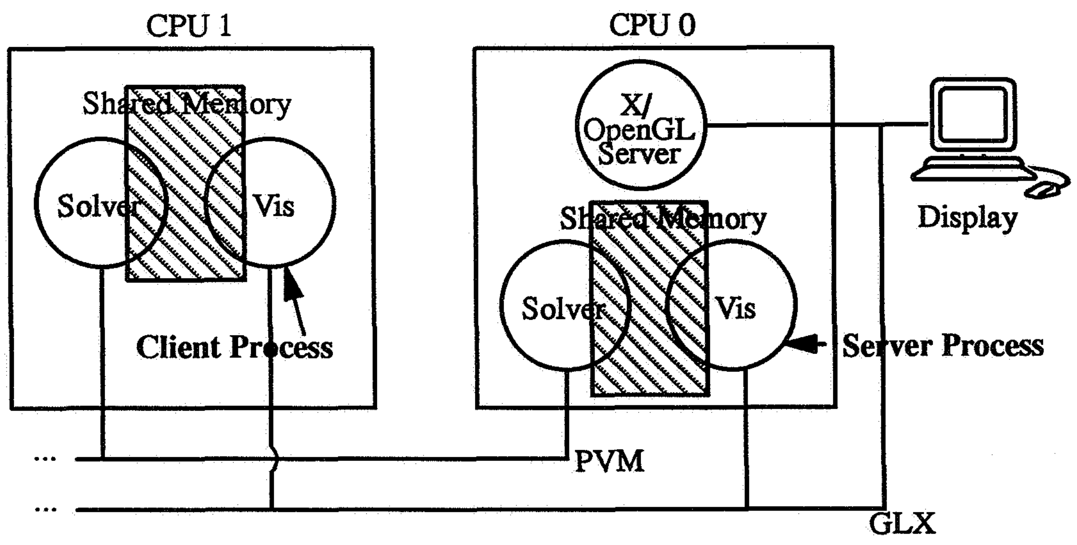

Figure 1. Architecture of the Simulation/Visualization System using OpenGL

In addition to providing a mechanism for remote rendering, a way was needed to communicate textual information between the server process and each client process. For example, what portion of each sub–domain does the user wish to see? Does the user wish to see velocity vectors or iso–surfaces of pressure? Each sub–domain creates it own separate window on the display for the purpose of extracting input information from the user. This information is feed straight back to the remote client. Early experimentation relied on textual information being passed to each field solver process which stored the information in shared memory. The shared memory was read by the visualization process when an *expose* event was received from the server process(this is discussed in more detail in the following section). This technique was abandoned because it required that the solver process be involved in the role of visualization, which violated a fundamental design goal.

With the process management in mind, the details of the graphics process will be explored in some detail.

Characterization of the OpenGL Library

OpenGL[7.] provides user control of the specification of such parameters as transformation matrices, lighting and fog coefficients, anti–aliasing methods, pixel operations, and others. It does not provide a format method for modeling the geometry of objects within the definition of OpenGL, thus it is not provide a descriptive definition. Instead, object definitions come from an ordered set of vertex and polygon calls which when combined with other calls that specify vertex color values or indeces, surface normals, and other attributes, defines how an object is rendered.

The execution model used under OpenGL, analogous to that of the X Window System, is characterized as client–server where an application(client) issues commands interpreted and processed by OpenGL(server). The current implementation of OpenGL under the UNIX derivative SGI operating system IRIX is integrated with the X11R5 server[4] through the GLX extension. The server may or may not operate on the same computer as a client, thus making OpenGL somewhat network transparent. The connection between the client and the server is represented in a GL *context*, each of which contains a unique rendering state within the server. With but a few exceptions, all of the *context* is stored on the server, and not the client. In effect, client side graphics calls are merely shuttled to the server for processing[5].

OpenGL supports both *immediate mode* and *display list* based rendering. *Display lists* are basically drawing instructions which are tokenized and cached, and indexed by a single integer identifier. A common use of the *display list* is to store graphic objects which will be drawn many times without change. *Display Lists* provide a convenient handle to graphic objects and eliminate the computational overhead of recalculation for complex objects. *Display Lists* play an another important role in distributed rendering in that *display lists* are cached on the server and therefore incur no penalty in terms of network bandwidth for local redraws on the server. However, the use of *display lists* incur a penalty of additional memory management on the server, therefore the chose of which mode to use must be tempered by the requirements of the application. Fortunately, OpenGL provides both mechanisms, and indeed, both methods may be appropriate within a single application.

To illustrate the sequence of events to creating a shared window, first a process is created on the workstation that sponsors the shared window. This process shall be referred to as the **server process** (See Figure 1). For client processes to share this window, it is only necessary for the remote process to know the name of the workstation that contains the shared window, and the *ID* of the *window*(or *pixmap*) that is to be shared. In OpenGL, the same information is used with the exception that the *window* ID is used to create a *GLXContext*(via the *XVisualInfo* structure). Afterward, any OpenGL rendering calls made by any of the client processes will be observed in the shared window. Of course, to render meaningful graphics, these events and resources must be synchronized between the **server process** and the client processes.

The GL *context*, being unique for each connection to the server, maintains separate Projection **P** and Model–view **MV** matrices(actually a matrix stack), among other resources. Classically, **P** is used to define the transformation from world coordinates into screen coordinates, and **MV** is used to transform from object space to the world coordinate system. **P** is constant during the construction of a scene and is necessary for all objects to be drawn in the proper relative position to each other on the display. Since each *context* can contain a non–unique **P**, some method is needed to synchronize **P** between each remote client process. For the current work, this is accomplished by defining a new X Window Property on the shared window called *glXProjectionMatrix*. Only the **server process** by convention is able to define P. This is done by establishing an X *Property* on the shared win-

4. OpenGL has also been implemented under Windows 3.1 and Windows NT using the Microsoft Windows API

5. OpenGL also supports the notion of direct rendering to the graphics hardware, thereby providing a performance advantage, but is implementation dependent and not used in the current work.

dow which contains the **P**. **P** is altered by the *server process* calling *XChangeProperty* which in turn generates *PropertyNotify* events in each client. Each client responds by calling *XGetWindowProperty* to retrieve **P** and issue the appropriate local calls to set the projection matrix for their respective GL *contexts*.

For single-buffered graphics, the client processes ignore native *expose* events, but chose to except *expose* events sent as a *client_message*. When the **server process** then chooses to redraw the window or receives an expose event from the window manager, it clears the shared window, then issues an *XSendEvent* with the core event type being an *expose*. The client processes receive this event and redraw everything requested within their sub-domain. The X Protocol does not specify that *selected* events be propagated to its clients in any certain order, therefore the "*faux-expose*" event is required to prevent any client from rendering into the shared window before it is cleared.

Double-buffered graphics is a bit more complex, but only slightly. The process is similar to single-buffering expect that the client processes are rendering into a back buffer which must be swapped with the front buffer when all the clients are finished drawing. But, how does the server process know when everyone is finished drawing? One solution is that if the **server process** is aware of how many clients are participating in the rendering, then each client can send a client event after the rendering is complete. When all the clients have responded, then the **server process** can swap the front and back buffers. Another perhaps more formal solution is to use the X Synchronization Extension[8.], version 2.0 for X11R5, or version 3.0 for X11R6.

Comparison to the Method of Extracts

In a nutshell, an *extract* can be characterized as a data structure containing geometric information. An *extract* is therefore descriptive, rather then procedural(see the Introduction), and represents a vastly different programming paradigm to the graphics programmer point-of-view. Logistically, however, an extract is analogous to a *display list* is the sense that both contain geometric information, and both can be used to represent graphic objects.

A distributed system using extracts might look like the following figure

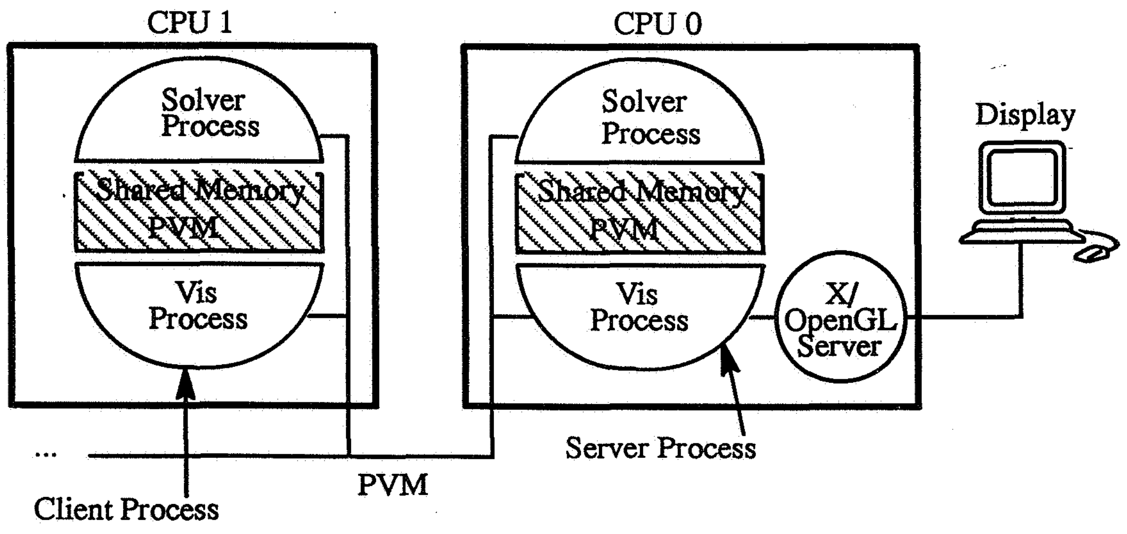

Figure 2. Architecture of a Simulation/Visualization System using extracts

with perhaps the most notable feature being that the solver and visualization process share a single run–time image of the message passing interface[6], and that only the **server process** renders to OpenGL.

Table 1 illustrates the division of labor for the client and server processes for each of the two models.

function	*Client Process*	*Server Process*	*OpenGL*
Extracts			
protocol definition	X	X	
transport	X	X	
rendering		X	X
display list caching	X	X	
Remote Rendering			
protocol definition			X
transport			X
rendering	X		X
display list caching			X

Table 1. Comparison of division of labor for the Method of Extracts and Remote Rendering

Clearly, Table 1 illustrates that the bulk of the burden for visualization using *extracts* lies within the end–user applications. What then would be the advantages of using extracts over remote rendering? An *extract* is created, managed, and transported at the user level within both the client processes as well as the **server process**. This gives a great deal of flexibility in how the *extract* can be managed(for example, it can be saved to file). There is no user–supported method in OpenGL for accessing the world coordinate vertex information in the display list.

The amount of data physically transported over the network for each method will be similar. Both are required to move vertex, normal, and other geometric data in floating point 32 and 64–bit format. Care must be exercised for the Method of Extracts if heterogeneous platforms are used to insure data consistency of the floating point format. OpenGL provides that support transparently to the user by using a network neutral floating point format.

As is almost always the case, the chose of which method is best depends on the application. If the primary motivation is to perform visualization and there is no need to retain the graphics in a polygonal format, then distributed rendering might be appropriate for your application. If the application performs post–processing of the graphics information or if retention of the world space polygonal data is required, then the Method of Extracts might be appropriate. Another option would be to use distributed rendering combined with a user–level display list which could be exchanged with the

6. Visual3 uses IRIX threads (sprocs) and therefore the solver process and visualization process inherently share an address space.

server process either via a *XProperty* or via the ICCCM[9.] mechanism in X Windows. This technique is analogous to combining distributed rendering with the Method of Extracts.

Results

The modified version of NPARC[10.] with distributed visualization was applied to a 4 block turbine engine configuration as shown in Figure 2. This problem was descretized into 4 unequal sized computational blocks(Figure 3) traversing the domain in the streamwise direction, and was run on 4 SGI Indigo's running under IRIX 5.0.1. A pressure plot of the resulting computation is shown in Figure 4. along a vertical plane for 6 chronological time frames taken during the resulting simulation. This solution was compared with the solution run with a single processor. The convergence within each block(Figure 5) was observed to be identical with the solution run in parallel indicating that decomposition did not negatively impact total run times. The single block case was run without visualization enabled indicating that the blocking within the solver process had little or no impact on the efficiency of the parallel environment.

The fundamental purpose of the current work was to understand the mechanisms and practical issues of using distributed visualization in a parallel environment for large scale simulations. The results are encouraging and are summarized in the following list:

1. Large scale CFD calculation with real–time non–obtrusive display of 3D field data is practical if the amount of requested data within each block is held to a reasonable level. It is important that the resources for visualization remain small compared with the simulation itself, otherwise it may be better to use the CPU for the simulation and use post–processing.
2. The ability to perform distributed rendering using OpenGL requires no modification to either the GLX extension or OpenGL. This capability was something SGI had in mind with the current design of the library.
3. Conceptually, remote rendering and the method of extracts offer no advantage over the other in terms of minimizing network bandwidth. Any advantages one method holds over the other will be application dependent.

Future

Many visual paradigms are fully scalable in the sense that work performed within each domain is totally independent of the other blocks(i.e. iso–surface construction, vector plots, etc). However, some tasks require cooperation between blocks, such as particle trajectory tracing. This brings in issues such as the scalability of the process, and how to exchange trajectory coordinates as particles migrate between domains. Any future work in parallel visualization must consider such issues.

Work will continue on building an integrated computational environment around the NPARC Navier–Stokes solver. The environment will provide the user the following capabilities:

1. Set and edit boundary and run–time parameters using a dynamic 3d graphics screen
2. View the computational grid
3. Submit the program and monitor the performance of the Parallel environment
4. Provide the ability to monitor the field solution in real–time as the field solution progresses using iso–surface, contour, vector plots, and others.

5. Stop the simulation, change parameters, then reset or resume processing.
6. The environment will support check-pointing and fault tolerance.

Longer term, if funding allows, support will be added for automated block decomposition to match the number of processors in the parallel processor. Ideally, integration with a grid generator would complete the tool-set needed for a complete computational environment.

References

[1.]. A. Globus, "A Software Model for Visualization of Time-Dependent 3-D Computational Fluid Dynamic Results," NASA Ames Research Center, NAS Systems Division, Applied Research Branch technical report RNR-92-031, November 1992.

[2.] G. Bancroft, F. Merritt, T. Plessel, P. Kelaita, R. McCabe, and A. Globus. "FAST: A Multi-Processing Environment for Visualization of CFD," *Proceedings Visualization '90,* IEEE Computer Society, San Franciso(1990).

[3.] P.G. Buning, J.L. Steger, "Graphics and Flow Visualization in Computational Fluid Dynamics," AIAA-85-1507-CP, AIAA 7th Computational Fluid Dynamics Conference, 15-17 July 1985, Cincinnati, OH.

[4.] CORBA, see ftp://omg.org:/pub.

[5.] NPARC, see http://info.arnold.af.mil/nparc/Solver_info.html#document.

[6.] PVM3, see http://www.netlib.org/pvm3/book/pvm-book.html.

[7.] OpenGL Reference Manual, Addison-Wesly Publishing Company, ISBN 0-201-63276-4, First Edition November 1992.

[8.] X Synchonization Extension, see ftp://ftp.x.org/pub/DOCS

[9.] R. Scheifler, James Getty, X Window System, Third Edition, Digital Press, ISBN 1-55558-088-2, 1992.

[10.] M.L. Stokes and D.H. Huddleston, "PANARC: Parallelized NPARC with 3D Real-Time Distributed Visualization," to be presented at the AIAA Joint Propulsion Conference, July 10-12, San Diego, California.

[11.] MPI, see http://www.erc.msstate.edu/mpi/#basic.